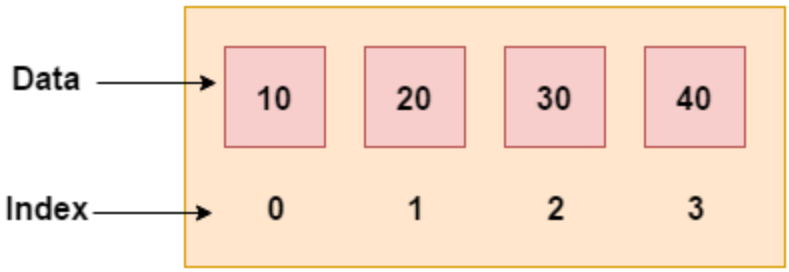
Chapter 4

Arrays and Functions

Arrays

Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location.

In C++ **std::array** is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.



Advantages of C++ Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

Disadvantages of C++ Array

- Fixed size

C++ Array Types

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array
2. Multidimensional Array

C++ Single Dimensional Array

Let's see a simple example of C++ array, where we are going to create, initialize and traverse array.

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5]={10, 0, 20, 0, 30};  //creating and initializing array
       //traversing array
       for (int i = 0; i < 5; i++)
       {
          cout<<arr[i]<<"\n";
       }
}
```

Output:

```
10
0
20
0
30
```

Following is an example, which will use declaration, assignment and accessing arrays :

```
#include <iostream>
using namespace std;

#include <iomanip>
using std::setw;

int main () {

   int n[ 10 ]; // n is an array of 10 integers

   // initialize elements of array n to 0
   for ( int i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; // set element at location i to i + 100
   }
   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output each array element's value
```

```
  for ( int j = 0; j < 10; j++ ) {
    cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
  }

  return 0;
}
```

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result −

```
Element       Value
      0         100
      1         101
      2         102
      3         103
      4         104
      5         105
      6         106
      7         107
      8         108
      9         109
```

C++ Array Example: Traversal using foreach loop

We can also traverse the array elements using *foreach* loop. It returns array element one by one.

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
       //traversing array
      for (int i: arr)
       {
          cout<<i<<"\n";
       }
}
```

Output:

```
10
20
30
```

```
40
50
```

C++ Multidimensional Arrays

In C++ you can define multidimensional arrays with any number of dimensions. The ANSI standard stipulates a minimum of 256 dimensions but the total number of dimensions is in fact limited by the amount of memory available. The most common multidimensional array type is the two-dimensional array, the so called *matrix*.

Example:

```
float number[3][10]; // 3 x 10 matrix
```

This defines a matrix called number that contains 3 *rows* and 10 *columns*. Each of the 30 (3 X 10) elements is a float type. The assignment

Example:

```
number[0][9] = 7.2; // Row 0, column 9
```

stores the value `7.2` in the last element of the first row.

Arrays as Array Elements

C++ does not need any special syntax to define multidimensional arrays. On the contrary, an n-dimensional array is no different than an array with only one dimension whose elements are (n–1)-dimensional arrays.

The array number thus contains the following three elements:

number[0] number[1] number[2].

Each of these elements is a float array with a size of 10, which in turn forms the rows of the two-dimensional array, number.

This means that the same rules apply to multidimensional arrays as to one-dimensional arrays. The initialization list of a two-dimensional array thus contains the values of the array elements, that is, the one-dimensional rows.

Examples:

```
int arr[2][3] = {
                    {5, 0, 0},
                    {7, 0, 0} };
int arr[][3] = { {5}, {7} };
```

These two definitions are equivalent. When you initialize an array, you can only omit the size of the first dimension. It is necessary to define any other dimensions since they define the size of array elements.

Let's see a simple example of multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

```
#include <iostream>
using namespace std;
int main()
{
  int test[3][3];  //declaration of 2D array
    test[0][0]=5;  //initialization
    test[0][1]=10;
    test[1][1]=15;
    test[1][2]=20;
    test[2][0]=30;
    test[2][2]=10;
    //traversal
    for(int i = 0; i < 3; ++i)
    {
        for(int j = 0; j < 3; ++j)
        {
            cout<< test[i][j]<<" ";
        }
        cout<<"\n"; //new line at each row
    }
    return 0;
}
```

Output:

```
5 10 0
0 15 20
30 0 10
```

Let's see a simple example of multidimensional array which initializes array at the time of declaration.

```
#include <iostream>
using namespace std;
int main()
{
```

```
  int test[3][3] =
    {
       {2, 5, 5},
       {4, 0, 3},
       {9, 1, 8}  };  //declaration and initialization
    //traversal
    for(int i = 0; i < 3; ++i)
    {
       for(int j = 0; j < 3; ++j)
       {
          cout<< test[i][j]<<" ";
       }
       cout<<"\n"; //new line at each row
    }
    return 0;
}
```

Output:"

```
2 5 5
4 0 3
9 1 8
```

C++ Functions

As we know that functions play an important role in C or C++ program development. Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

The function in C++ language is also known as procedure or subroutine in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability.

Recall that we have used syntax similar to the following in developing C programs.

```
void show();   /* Function declaration */
int main()
{
    …….
    show() ;       /* Function call */
    …….
}
void show()    /* Function definition  */
 {
   …….
 }
```

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are executed and control returns to the main program when the closing brace is encountered. C++ is no exception. Functions continues to be the building blocks of C++ programs. In fact, C++ has added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ function can be overloaded to make it perform different tasks depending on the arguments passed to it. Most of these modification are aimed at meeting the requirements of object oriented facilities.

A function in C++ is a group of statements that together perform a specific task. Every C/C++ program has at least one function that the name is main. The main function is **called by the operating system by which our code is executed**.

We can make n number of function in a single program but we can make only one **main function** in a single program. Every program has only one main function.

The main part of functions is **return_type, function_name, parameter and functions body.**

Syntax :-

```
return_type Function_name (Parameters)
{
    // Function Body
}
```

- **Return Type** − A function may return some value. A **return_type** is the data type of the value the function returns. Some functions perform the desired operations and do not return any value. In this case, the return_type is specified by the keyword **void**.
- **Function Name** − It is the actual name of the function. The function name and the parameter list together form the function signature.
- **Parameters** − A parameter is like a placeholder. Whenever a function is called or invoked, you pass a value to the parameter. This value is referred to as an actual parameter or argument. The parameter list refers to the type, order, and a number of the

parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Function Body** − The function body constitutes a collection of statements that define what the function does.

Advantage of functions in C++

There are many advantages of functions.

1) Code Reusability : By creating a functions in C++, you can call it many times. So we don't need to write the same code again and again.

2) Code optimization

It makes the code optimized, we don't need to write much code.

Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

Function Declaration

Before writing the body of function just declare the function. if we declare the function then we write the definition of function anywhere in the program. We always declare a function above the main function. In a function declaration, we use; in last.
Syntax-

```
return_type Function_name(arguments);
 Function Definition
```

Function definition show the work of function or this is define what function do.
Syntax-

```
Return_type Function_name(arguments)
{//body}
```

How to call a Functions

Four ways to call a functions
Syntax-

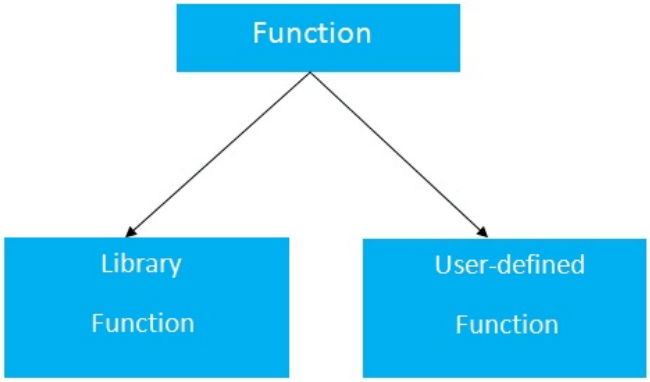
```
1  Function_name();
2   Function_name(arguments);
3  Variable_name = Function_name();
4  Variable_name = Function_name(arguments);
```

Types of functions

Broadly functions are categorized into two category:

1. Library Functions: are the functions which are declared in the C++ header files such as ceil(x), cos(x), exp(x), etc.

2. User-defined functions: are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.



We can further declare a function in four different ways:-

1. A function that not take any argument and not return any value that type of function is take nothing and return nothing type

Syntax:-

```
void  Function_name (void)
{
   // Function Body
}
```

A function that takes an argument and not returns any value that type of function is take something and return nothing type

Syntax:-

```
void  Function_name (Parameter)
{
   // Function Body
}
```

A function that not take any argument and return any value that type of function is take nothing and return something type

Syntax :-

```
Return_type Function_name (void)
{
   // Function Body
}
```

A function that takes an argument and return any value that type of function is take something and return something type

Syntax :-

```
Return_type  Function_name (Parameter)
{
   // Function Body
}
```

Let's understand this concept with one example

Here we take the addition of two number as an example.

Take Nothing Return Nothing type function

```
 int main()——————————line 1
{
  addition();——————————line 2
}
```

line1- Simply we declare the main function in every program.
Line2- Here we call the function that doesn't take any arguments and return nothing.

Let's see the body of addition function

```
 void addition( void ) ————LINEA
{
    int a=10,b=20,c;————line3
    c=a+b;————line4
   cout<<c; ————line5
}
```

Line3- here we initialize a and b with 10 and 20 and declare a variable name c.
Line4- we store the addition of a and b in c
Line5- it shows the addition of a and b or prints the value of c.

Take Something Return Nothing type function

```
 int main()——————————line 1
{
   int a=10,b=20;——————————line 2
  addition(a,b);——————————line 3
}
```

line1- Simply we declare the main function in every program.
line2- we initialize two variable that is int type the value of a is 10 and b is 20.
line3- Here we call the function that takes two arguments and returns nothing.

Let's see the body of addition function

```
 void addition(int a, int b) ————LINEA
{
   int c;   ————line4
   c=a+b;   ————line5
   cout<<c;   ————line6
}
```

line4- here we declare a variable name c.
line5- we store the addition of a and b in the c
line6- it shows the addition of a and b.
Note:- a and b of both line (**LINE A and line 3)** are different. We can take another variable name on **LINE A.**

Take Nothing Return Something type function

```
 int main()——————————line 1
{
    int c;——————————line 2
    c=addition();——————————line 3
    cout<<c;——————————line 4
}
```

line1- Simply we declare the main function in every program.
line2- we initialize the c variable that store the function return value.
line3- Here we call the function that takes nothing but returns some value that is store inc.
Line4 – print the value that function **addition** return.

Let's see the body of addition function

```
 int addition( void ) ————LINEA
{
    int a=10,b=20,c;————line5
   c=a+b;————line6
   return c————line7
}
```

Line5- here we declare a variable name c.
Line6- we store the addition of a and b in c
Line7- it returns the c.

Take Something Return Something type function

```
 int main()——————————line 1
{
    int a=10,b=20 , c ; ——————————line 2
   c=addition(a,b); ——————————line 3
   cout<<c;——————————line 4
}
```

line1- Simply we declare the main function in every program.
line2- we initialize a,b and declare c.
line3- Here we call the function that takes and b as arguments and return some value that is store inc.
Line4 – print the value that function **addition** return.

Let's see the body of addition function

```
 int addition( int m, int n) ————LINEA
{
int c;————line5
c=n+m;————line6
return c————line7
}
```

Line5- here we declare a variable name c.
Line6- we store the addition of m and n in c
Line7- it returns the value of c.

Let's see the simple example of C++ function using static variable.

```
#include <iostream>
using namespace std;
void func()
{
  static int i=0; //static variable
  int j=0; //local variable
  i++;
  j++;
  cout<<"i=" << i<<" and j=" <<j<<endl;
```

```
}
int main()
{
 func();
 func();
 func();
}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

Call by value and call by reference in C++

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.

Let's understand call by value and call by reference in C++ language one by one.

Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Let's try to understand the concept of call by value in C++ language by the example given below:

```
#include <iostream>
using namespace std;
void change(int data);
int main()
{
int data = 3;
change(data);
cout << "Value of the data is: " << data<< endl;
return 0;
```

```
}
void change(int data)
{
data = 5;
}
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in C++ language by the example given below:

```
#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
 int swap;
 swap=*x;
 *x=*y;
 *y=swap;
}
int main()
{
 int x=500, y=100;
 swap(&x, &y);  // passing value to function
 cout<<"Value of x is: "<<x<<endl;
 cout<<"Value of y is: "<<y<<endl;
 return 0;
```

```
}
```

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

C++ Passing Array to Function

In C++, to reuse the array logic, we can create function. To pass array to function in C++, we need to provide only array name.

```
functionname(arrayname); //passing array to function
```

Let's see an example of C++ function which prints the array elements.

```
#include <iostream>
using namespace std;
void printArray(int arr[5]);
int main()
{
    int arr1[5] = { 10, 20, 30, 40, 50 };
    int arr2[5] = { 5, 15, 25, 35, 45 };
    printArray(arr1); //passing array to function
    printArray(arr2);
}
```

```
void printArray(int arr[5])
{
   cout << "Printing array elements:"<< endl;
   for (int i = 0; i < 5; i++)
   {
            cout<<arr[i]<<"\n";
   }
}
```

Output:

```
Printing array elements:
10
20
30
40
50
Printing array elements:
5
15
25
35
45
```

Default arguments in C++

In a function, arguments are defined as the values passed when a function is called. Values passed are the source, and the receiving function is the destination.

Now let us understand the concept of default arguments in detail.

A default argument is a value in the function declaration automatically assigned by the compiler if the calling function does not pass any value to that argument.

Characteristics for defining the default arguments

Following are the rules of declaring default arguments -

- The values passed in the default arguments are not constant. These values can be overwritten if the value is passed to the function. If not, the previously declared value retains.
- During the calling of function, the values are copied from left to right.
- All the values that will be given default value will be on the right.

Example

```
void function(int x, int y, int z = 0)
```

Explanation - The above function is valid. Here z is the value that is predefined as a part of the default argument.

```
void function(int x, int z = 0, int y)
```

Explanation - The above function is invalid. Here z is the value defined in between, and it is not accepted.

```
#include<iostream>
using namespace std;
int sum(int x, int y, int z=0, int w=0) // Here there are two values in the default arguments
{ // Both z and w are initialised to zero
   return (x + y + z + w); // return sum of all parameter values
}
int main()
{
   cout << sum(10, 15) << endl; // x = 10, y = 15, z = 0, w = 0
   cout << sum(10, 15, 25) << endl; // x = 10, y = 15, z = 25, w = 0
   cout << sum(10, 15, 25, 30) << endl; // x = 10, y = 15, z = 25, w = 30
   return 0;
}
```

Output

```
25
50
80
```

In the above program, we have called the sum function three times.

- Sum(10,15)
  When this function is called, it reaches the definition of the sum. There it initializes x to 10 y to 15, and the rest values are zero by default as no value is passed. And all the values after sum give 25 as output.
- Sum(10, 15, 25)
  When this function is called, x remains 10, y remains 15, the third parameter z that is passed is initialized to 25 instead of zero. And the last value remains 0. The sum of x, y, z, w, is 50 which is returned as output.

- Sum(10, 15, 25, 30)
  In this function call, there are four parameter values passed into the function with x as 10, y as 15, z is 25, and w as 30. All the values are then summed up to give 80 as the output.

> *Note If the function is overloaded with different data types that also contain the default arguments, it may result in an ambiguous match, which results in an error.*

```
#include<iostream>
using namespace std;
int sum(int x, int y, int z=0, int w=0) // Here there are two values in the default arguments
{ // Both z and w are initialised to zero
   return (x + y + z + w); // return sum of all parameter values
}
int sum(int x, int y, float z=0, float w=0) // Here sum is overloaded with two float parameter val
ues
{ // This results in ambiguity
   return (x + y + z + w);
}
int main()
{
   cout << sum(10, 15) << endl; // x = 10, y = 15, z = 0, w = 0
   cout << sum(10, 15, 25) << endl; // x = 10, y = 15, z = 25, w = 0
   cout << sum(10, 15, 25, 30) << endl; // x = 10, y = 15, z = 25, w = 30
   return 0;
}
```

Output

```
prog.cpp: In function 'int main()':
prog.cpp:15:20: error: call of overloaded 'sum(int, int)' is ambiguous
  cout << sum(10, 15) << endl; // x = 10, y = 15, z = 0, w = 0
                    ^
prog.cpp:4:5: note: candidate: int sum(int, int, int, int)
 int sum(int x, int y, int z=0, int w=0) // Here there are two values in the default arguments
     ^
prog.cpp:9:5: note: candidate: int sum(int, int, float, float)
 int sum(int x, int y, float z=0, float w=0) // Here sum is overloaded with two float
```

Here when we call the sum function with all the **parameters(x, y, z, w)** or either any one parameter value of z or w, the compiler gets confused about which function to execute. Thus, it creates an ambiguity which results in the error.

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types

```
#include <iostream>
using namespace std;

class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};

int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print float
   pd.print(500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Printing int: 5
Printing float: 500.263
Printing character: Hello C++

Output:

```
Here is int 10

Here is float 10.1

Here is char* ten
```

How Function Overloading Work?

- *Exact match*:- (Function name and Parameter)
- *If* a *not exact match is found:–*
  ->Char, Unsigned char, and short are promoted to an int.

  ->Float is promoted to double
- *If no match found*:
  ->C++ tries to find a match through the standard conversion.

Inline Function

Inline function is one of the important feature of C++. So, let's first understand why inline functions are used and what is the purpose of inline function?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
   // function code
```

}

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

1) If a function contains a loop. (for, while, do-while)
2) If a function contains static variables.
3) If a function is recursive.
4) If a function return type is other than void, and the return statement doesn't exist in function body.
5) If a function contains *switch* or *goto* statement.

The following program demonstrates the use of use of inline function.

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
   return s*s*s;
}
int main()
{
   cout << "The cube of 3 is: " << cube(3) << "\n";
   return 0;
} //Output: The cube of 3 is: 27
```